Express Mail Label: EL761869465US
Date of Deposit: May 11, 2001

Scalable Processor

Description

Technical Field

This invention related to a method and apparatus for issuing and executing instructions in a computer system.

Background of the Invention

A significant portion of the complexity of current processors can be attributed to their attempts to mask the latency of memory accesses. Multi-threading, out of order processing, prefetching memory data, speculative executions are all examples of this. Technology trends indicate that memory speeds are unlikely to catch up with processor speeds. While current memory designs exhibit limited facilities of pipe lining and hierarchy, we have shown in a co-pending application the design of a scalable pipeline hierarchy which gives a linear latency function at constant bandwidth. The co-pending application, IBM Docket YOR920010439US1, is being filed concurrently with the instant application and entitled, "Scalable Memory" and is incorporated herein by reference. However, current day processors cannot exploit such an unbounded pipeline as they tend to remember outstanding memory requests. Since they can have only finite resources, they can exploit the memory pipeline to a very limited extent. The resources include a finite number of buffers to store information about an instruction associated with a tag. For instance, IBM PowerPC processors can have at most 8 to 16 outstanding memory operations and other competing processors have even lower limits. This limit exists because a processor has dedicated resources to remember pending memory requests, and to indicate further processing of the data after it arrives. For example, when a response to an issued memory instruction is returned from the memory of the computer system, the response would include only data retrieved from memory and the memory tag. In order to execute the issued instruction, the tag is used to retrieve the op code and the target address which is stored in buffers the processor. Another source of limitation is the finite number of registers in the processor and the processor's inability to operate

on any data that is not present in the registers. In today's processors, instructions can only be executed with operands present in the registers of the processor, so the number of registers imposes a limitation of the number of instructions that can be executed concurrently.

Summary of the Invention

It is an aspect of this invention to construct a processor that can issue unbounded number of memory requests that can be processed in a pipe-lined manner. This aspect is accomplished by organizing all instructions to manipulate data from memory and packaging memory requests with sufficient information to process the returned data independently. This aspect separates the instruction issue and execution components of a processor, and each of them maintains very minimal state information and operates in an almost "stateless" manner. The minimal state information includes a program counter and a few fence counters.

It is another aspect of this invention to avoid reading data from a memory before a corresponding write to memory, that is to avoid what is known as the read-after-write hazard. This aspect is particularly important when a large number of instructions are issued from the processor, where all of the issued instructions have not been executed, that is when there are a large number of outstanding writes to memory that have not been completed. Such a large number is possible when using the linear memory described in the previously mentioned co-pending application, which is assigned to the same assignee as that of the instant application.

This invention avoids the read-after-write hazard by maintaining a fence counter or counter for each of a number of regions of the memory, where the counter value is used to control the issuance of further instructions from a processor. According to this invention , when a fence instruction designating a particular memory region is issued, no further instructions will be issued if the counter for the particular memory region is above a threshold. The counter for each memory region is incremented each time an instruction, whose target location is in that region, is issued and decremented each time an instruction is executed writing to that memory region. The threshold value is typically set to zero.

Brief Description of the Drawings

FIG. 1 is a schematic illustration of the scalable processor in accordance with this invention.

FIG. 2 schematically illustrates the read after write hazard which occurs when a read operation is submitted before a corresponding write operation.

FIG. 3 is a schematic of a linear memory system that could be used with the processor of this invention.

FIG. 4 is a flow chart of the logic used in the controllers for forwarding responses to requests to the processor.

FIG. 5 is a state diagram, where each state indicates the number of responses stored in the buffers of a controller d_i

Detailed Description of the Invention

FIG. 1 illustrates the general scheme of the scalable processor system 100. It uses the linear memory 16 described in the previously mentioned IBM Docket YOR920010XXX. For convenience, we have extended the memory to have one port 3 for read requests and two ports 4 for write requests. This is done purely to match the linear memory 16 which can deliver two results in each cycle. Two execution units 17 consume these results 14 and submit write requests independently. The organization ensures that requests on these lines 12 do not interfere with each other.

The upward path is divided into three concurrent paths 11-12, one path 11 carrying the read requests to the memory, the other two paths 12 carrying write requests to the memory. This is purely a convenience and does not increase the input bandwidth in any manner.

The read request carries two target addresses x and y. It first travels to the earlier location, collects the data and travels further up on the upward path to the second location and collects the second piece of data. The result travels on the downward path 13 carrying the pair of data items requested. This results a constant increase in the latency on the paths, but does not change the bandwidth considerations in any manner.

Instruction Format:

The processor, will have no registers and uses a linear memory hierarchy for all its data - the lowest level representing registers. The general form of an instruction is op(x,y,z) where x,y,z are

addresses of memory locations, and the semantics is to perform the operation (op) on the data from locations x and y and store the result into location z. The second operand may be omitted for unary operations. A unary instruction frequently used is the move(x,z) instruction, which copies the data from location x to location z.

Instruction Issue and Execution:

Referring to FIG.1, a processor 9 consists of a single issue unit 19 that issues instructions 50 and two execution units 17, each of which can perform any operation when the data is available. Several issue and execution units can be implemented in hardware on a single hardware chip. An instruction goes through two phases: Issue phase and Execution phase. In the Issue phase, a dual-operand fetch is submitted to the linear memory 10. The op code and destination information are attached to the request and returned with the data. The execution phase starts when the memory returns the dual-operand data. Each execution unit receives operand data 14, performs an operation and submits a write request to the memory 10 to store the result. Instructions are issued in the same order in which they are listed in the program. Since the memory does not do any data movement other than what is specified by the instructions, the programs have full control of the memory and do the memory management explicitly by issuing move instructions to bring data closer to the processor as and when they are needed. The pipe lined nature of the memory facilitates concurrent movement of data while executing other instructions. Typically, using this invention, a programmer would use move instructions to bring data closer to the processor by the time the data is needed by the processor.

Read-after-write Hazards:

Since the Issue unit and the Execution units operate concurrently, one must ensure that a write operation to a location is submitted to the memory before a corresponding read operation to that location is submitted. See FIG. 2 for an illustration. For an instruction op(x,y,z), the write request to z is said to be outstanding during the time interval between issuing and executing that

instruction. Hazards are avoided by keeping track of outstanding writes to regions of memory as described below. In FIG. 2, the path of solid lines illustrates how the move instruction 21 is executed. When it is issued, location y is read and the data flows into the execution unit 17. When it executes, the result is sent to be stored in location x. However, the issue unit proceeds concurrently and issues other instructions following it. The add instruction 22 is an example of a subsequent instruction that uses x and its path is illustrated by the dashed line 11a. If this add instruction is issued before the previous store to x takes place, we have a hazard.

Fence Counter:

The processor is equipped with a finite set of fence counters. Each fence counter is associated with a region of memory. By choosing the sizes of the regions as powers of 2, we need log N, and, thus, maintaining log N counters is not a serious limitation in practice. Referring to Fig. 3, each time an instruction such as op(x,y,z) is issued 31, the fence counter associated, with the range containing z is incremental 36, and a request is sent to memory to read the data at locations, for example, x and y. See 37 of FIG. 3. The data retrieved (data 1, data 2), for example, is then forwarded to the execution unit, as shown in 38 of FIG. 3. When the execution unit completes that operation 33, i.e.. op (data 1, data 2, z), and submits a write request to location z, 33, the corresponding fence counter (See 18 of FIG. 1) is decremented 34. Thus, the fence counter contains the number of outstanding writes to locations in its associated range. Fencing is accomplished explicitly by the program (*a la* release consistency model) by inserting a special instruction, in the form of: fence (region of memory), where appropriate. See 23 of FIG. 2. When a fence instruction is encountered by the issue unit 32, the issue unit stalls 35 until the corresponding fence counter becomes zero. This mechanism can be used by programs to avoid read-write hazards and can be made efficient by choosing the ranges and placement for the fence operations in a judicious manner. Typically, a counter is implemental in hardware.

The extreme case is to have every operation succeeded by a fence on its target location. While this works correctly, the execution is nearly sequential. A program can be organized to accumulate as many instructions as possible that write into a region and then a fence instruction is posted before accessing any location from that region. Compiler technology can be developed to

judiciously choose the positions for fence operations.

FIG. 4 illustrates the general scheme of the scalable processor system 100. It uses the linear memory 16. For convenience, we have extended the memory to have one port 3 for read requests and two ports 4 for write requests. This is done purely to match the linear memory 10 which can deliver two results in each cycle. Two execution units 17 consume these results 14 and submit write requests independently. The organization ensures that requests on these lines 12 do not interfere with each other.

The read request carries two target addresses x and y. It first travels to the earlier location, collects the data and travels further up on the upward path to the second location and collects the second piece of data. The result travels on the downward path 13 carrying the pair of data items requested.

The structure of the proposed memory system 10 is shown in FIG. 4. For ease of illustration, the system is shown to have a sequence of building blocks laid out in one dimension. The blocks are numbered starting from L_1 at the processor and increasing as we go away from the processor. Each i-th building block L_i has unit memory, denoted by m_i, and two controllers, u_i and d_i. All the controllers u_i are connected by single links 5 forming the "upward path" carrying requests from the processor to the memory cells. The controllers d_i are connected by pairs of links 7 forming the "return path" and carry responses from memory cells to the processor. The design can sustain one request per unit time from the processor on the upward path, but the processor is required to be able to receive up to 2 responses in one unit time along the return path. Thus, this design requires an output bandwidth that is twice that of the input bandwidth. Also shown is a single processor 9 connected to this memory system. A processor such as that described in a related application, which is being filed concurrently with and by the assignee of the instant application, could be used with the memory system of this invention. A memory request specifies a target memory cell address, the operation (read/write) and data if it is a write operation. For a memory of size n, the target address is any integer between 1 and n. Any number greater than n can be used to simulate a no-operation (i.e. the processor did not submit any real request).

Requests and responses also carry additional information that is not interpreted by memory. For instance, if the processor executes an instruction of the form op(x,z) where x,z, are addresses of memory locations, the semantics is to perform the operation (op) on the data from location x of memory and to store the result into location z of memory. For this instruction, the memory request submitted is of the form [read, x, no-data, <op,z>]. The response to this request is of the form [read, x, data, <op,z>] where data is the information that was retrieved from location x. This response is forwarded on the return path through a series of second controllers to the processor. In the sample shown, when the response is received by the processor, the operation is performed on data to obtain a result, say w. Another request is then forwarded from processor through the first controllers on the upward path to store the result w at memory location z. The format of this request may look something like [write, z, w, no-info], which means store the value w at location z.

Each request with target address i travels on the upward path and a copy of it reaches the memory unit at every level. The memory unit m_i reads or stores the data depending upon the operation and forwards it to the controller d_i. A write operation terminates here without generating any response, whereas a read operation causes the request to be converted into a response which travels along the return path until it reaches the processor. If we assume unit times to travel the communication links and unit time for reading the memory, a read request for target address i takes 2i+1 units of time in the absence of any congestion during its travel. The controllers are designed to deal with congestion and ensure the flow of requests or responses subject to the requirements stated in the preceding section. Referring to FIG. 5, the detailed logic for u_i, m_i and d_i are specified below:

Operation in each cycle:

See FIG. 4.

- For each request received by u_i, one copy is sent to m_i and another copy is sent to u_{i+1}. At the top (when i is n) the second copy is simply discarded.
- m_i always forwards the request to d_i, after copying data from memory into request for a read operation, or copying data from request into memory for a write operation.

- As shown in FIG. 4, d_i has three internal buffers 6 which are organized as a FIFO queue. At the beginning of each cycle, d_i transfers any responses to requests present on the 2 links from d_{i+1} into its internal buffers. Then, the following algorithm (See flow chart in FIG. 5) is used to put responses on the two outgoing links to d_{i-1}:.

  1 If request from m_i is a read to location i, then it is converted to a response and is placed on the outgoing link. In addition, one response from the internal buffers of d_i (if any) is removed and placed on the outgoing links. (See 51 of FIG. 5.)

  2 If request from m_i is a write to location i(52), or the request is targeted to a higher location, then up to two responses from the internal buffers of d_i (if any) are removed and placed on the outgoing links (54).

  3 If request from 3 m_i is to a lower location (55), then 1 response from the internal buffers of d_i (if any) is removed and placed on the outgoing links (56).

Properties of the Model:

We now show the two properties required for scalability: constant buffer size in each unit and linear access latency:

- The size of the internal buffer of any d_i will never exceed 3. FIG. 6 shows the possible transitions for x_i, which is the number of filled buffers in d_i after each cycle. The invariant for the state x_i=3 asserts that in that state at most one response can come in through the incoming links and this ensures that x_i will never exceed 3. To show the invariant, we examine the 2 possible transitions into the state x_i=3: Consider the first transition to state x_i=3. This transition occurs when initially x_i=2 and both the two incoming arcs on the link from d_{i+1} (See 7 of FIG. 4.) carry responses and the request to m_i is to location i or lower. This ensures that in the next cycle, there can be at most one response over the incoming arcs from d_{i+1}. This is because m_{i+1} will process in the next cycle, a copy of the same response that m_i processed in this cycle and hence d_{i+1} will do case 3 of the above algorithm, outputting only one response. (See 55 and 56 of FIG. 5.) Now consider the second transition to state x_i=3. This transition occurs when initially x_i=3, there was one incoming request from d_{i+1}, and the request to m_i is to location i or lower. This again ensures that in the next cycle, there can be at most one response over

the incoming arcs from d_{m}.

- A read request to location i will return a corresponding response to the processor within 4i+1 cycles. The response to the read request reaches d_i after i+1 cycles since its path length is i+1 and there are no delays on these paths. The controller d_i immediately puts it on the outgoing arc as it executes case 1 of the algorithm. According to the algorithm, all buffers are emptied in FIFO order and at least one response from a buffer is removed in every cycle. Consequently, the response from d_i can experience a maximum delay of 3i units before it reaches the processor. Hence the total delay is at most 4i+1 for forwarding a request on the upward path and the corresponding response on the return path to the processor.

We observe that the design preserves the order of memory operations at each location while the order of completion of operations on different locations is unspecified. The proposed memory system can be implemented using standard memory logic which is incorporated into memory chips.